# Modern szoftverfejlesztési elvek és módszerek

Dr. Ekler Péter
Egyetemi adjunktus
BME VIK AUT

Balogh Tamás
Vezető fejlesztő
AUTSOFT

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

# Tartalom

- Bevezetés
- SOLID Tervezési elvek
- Clean Code – Szoftverfejlesztési szabályok
- Refactorálás
- Test Driven Development fejlesztési elvek

# Bevezetés

# Kihívások

- Volt már valaha dolgunk rossz kóddal?
  - „Olvasni a kódot több idő, mint írni”
- Volt már olyan eset, hogy nem volt idő egy feladat „szakszerű” megoldására, a kód tisztítására, a rövid határidő miatt?

„Ahogy nő a kód mennyisége, úgy csökken a fejlesztők produktivitása.”

# Code rot (kód romlás)

- Az alkalmazások tipikusan letisztult, tiszta architektúrával indulnak
- Mi történik egy bizonyos idő után?
  - A kód elkezd „rothadni” (romlani): kicsi hack itt-ott, egyre több *if* elágazás, mígnem az egész kódban ezek dominálnak -> átláthatatlan viselkedés
- Nehéz karbantartani, nehéz új funkciókat hozzáadni -> a fejlesztők egy idő után áttervezésért könyörögnek

# Kód romlás

- A forráskód bizonyos szempontból a terv (design)
- A romlott design és a rossz architektúra tipikus tünetei
  - Merevség
    - Folyamatosan nehezebb a kód módosítás
    - A változtatás költsége magas
  - Törékenység
    - Apró változtatás egy modulon egy másik modulban okozhat hibás viselkedést
    - Például: egy bug javítás elront egy látszólag független részt
  - Mozdulatlanság
    - Egy rendszer mozdulatlan, ha a részeit nem lehet könnyedén modulokba kiszervezni és máshol újra hasznosítani
    - Például: a login modul újra felhasználható legyen
    - Mozdulatlanság elkerülési stratégiák: rétegek kialakítása (adatbázis és UI különválasztása)
  - Nyúlékonyság
    - A kód struktúra nyúlékonysága
      - Új feature implementálását könnyebb megoldani hackeléssel, mint új kód írásával/új osztály bevezetésével
    - A környezet nyúlékonysága
      - Fordítás, teszt futtatás és becheckolás körülményes és sok ideig tart

# Kód romlás – Mi az okozója?

- Változó követelmények

  -> Ha olyan a kódunk/architektúránk, hogy nehéz a változásokat kezelni, az a mi hibánk

  -> A kód/architektúra rugalmas kell legyen a változások követésére és meg kell akadályozza a kód romlást

- Milyen változások miatt kezd romlani a kód? *Olyan változások, amelyek új, nem tervezett dolgokat hoznak az osztály függőségek szintjén.*
- A legtöbb tünet direkt, vagy indirekt módon a modulok közti nem megfelelő függőségre vezethető vissza.
- Az objektum orientált tervezési elvek segítenek a modulok közti függőségek kezelésében.

# A szoftver két értéke

- **Másodlagos érték**: a szoftver viselkedése - a szoftver azt csinálja hibamentesen, amit a felhasználó elvár
- **Elsődleges érték**: Tolerálja és egyszerűen alkalmazkodik a folyamatos változásokhoz, tehát könnyű módosítani (software is soft)

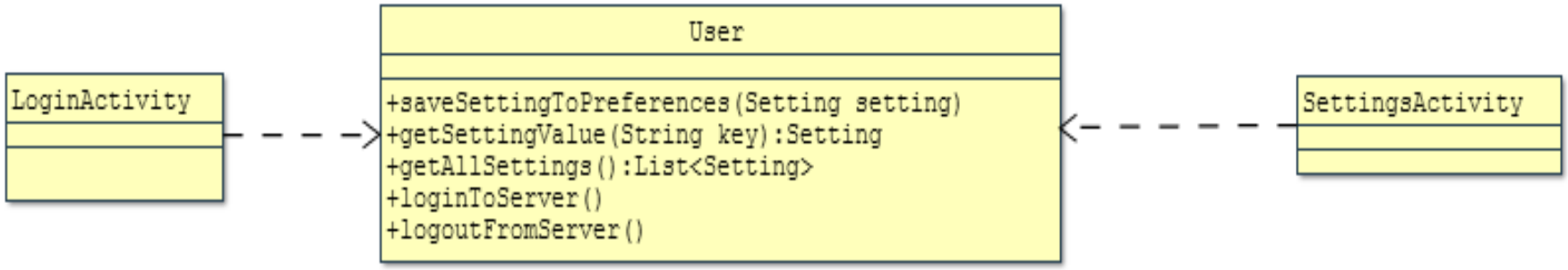
# SOLID Tervezési elvek

# Függőség kezelő OO elvek: SOLID elvek

- Single Responsibility Principle (S.R.P.)
- Open Closed Principle (O.C.P.)
- Liskov Substitution Principle (L.S.P.)
- Interface Segregation Principle (I.S.P.)
- Dependency Inversion Principle (D.I.P.)

# Single Responsibility Principle

- *Egy osztályt csak egy ok miatt módosítsunk*
- Példa az elv megszegésére: Mennyi okunk lehet a User osztályba való változtatásokra? (Másképp: mennyi felelőssége van a User osztálynak?)



# Single Responsibility Principle

- Mit jelent a felelősség?
  - Egy ok a változtatásra, a változás forrása: ha több mint egy ok lehet az osztály módosítására, akkor az osztálynak több mint egy felelőssége van
  - Az előző User felelősségei: beállítások, authentikáció
  - Ha ezt az elvet megsértjük, **törékeny** lesz az architektúra
  - Egy osztálynak egy „felhasználója” legyen

# Single Responsibility Principle további gondolatok

- Fölösleges komplexitás
  - Fel kell-e készítsük a kódot a jövőbeni esetleges sokfajta lehetséges követelményre?
  - Az architektúra az aktuális követelményekre koncentráljon
- Kohézió
  - Egy osztály kohéziója akkor maximális, ha minden metódusa használja az összes tagváltozót

# Open Closed Principle

- Definíció: **egy modul nyitott kell legyen a továbbfejlesztésre (könnyű a viselkedést kiterjeszteni), de zárt a módosításra (új dolog fejlesztéséhez nem kell a régi kódhoz nyúlni)**
- Következmény: új funkció hozzáadásakor új kód készül és nem a meglévő módosul
- Lehetséges ez?
  - Elméletben igen…
  - A probléma, hogy tipikusan van egy fő életciklus ami miatt a teljes rendszerre nem érvényesül az OCP, de kisebb részekre érvényesülhet, úgymint modulok, osztályok

# Liskov Substitution Principle

- Mi az a típus?
  - Nem számít, hogy mi a mögöttes megvalósítás
  - Csak a típuson végezhető műveletek számítanak

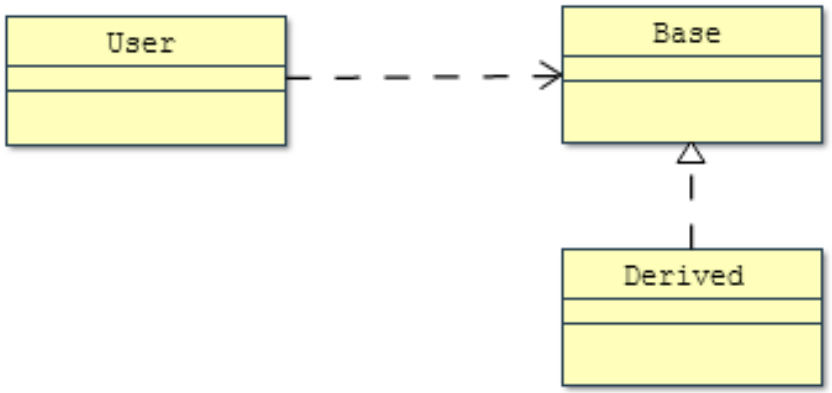
  -> *A típus csak a műveletek egy halmaza (a mögöttes struktúra rejtett)*
  - Példa:

```
public class Point {

  public int x;

  public int y;

}
```

    - Ez nem egy típus, csak egy adat struktúra, mivel nincsenek műveletei
    - Típussá alakítható műveletek hozzáadásával

# Liskov Substitution Principle

- Mit értünk altípus alatt?
  - *„S a T altípusa, ha minden S típusú o1 objektumhoz létezik T típusú o2 objektum úgy, hogy minden P programra a T-re értelmezve, a P viselkedése változatlan amikor o2 helyettesítve van o1-el.”* - *Barbara Liskov*
  - A leszármazott osztály helyettese lehet az ős osztálynak: Ha a User osztálynak egy függvényének van egy Base típusú paramétere akkor a helyére Derived típusú objektumot is átadhatunk és nem változik a viselkedés.



# Liskov Substitution Principle

- Visszautasított örökség
  - Ha az altípus függvénye mást csinál mint amit a használója elvár (az ős viselkedése alapján), vagy van mellékhatása
- A reprezentánsok szabálya
  - A valós életbeni objektumok közti viszonyokat nem kell feltétlenül megtartani a megvalósítás során
  - Például: a téglalap a négyzet leszármazottja?

# Interface Segregation Principle

- ***Ne függjön az osztály olyan dolgoktól, amire nincs szüksége,*** egyébként merev és törékeny lesz a megoldásunk.
- Ne kényszerítsük, hogy olyantól függjenek a felhasználók, amire nincs szükségük.
- Az elv egyik célja a „kövér” és „szennyezett” interfészek kiküszöbölése.
- A kövér interfészek rossz kohézióra utalnak. Más szóval az osztály interfészei szétválaszthatók függvény csoportokra.
- Jól kapcsolódik a Single Responsibility elvhez. Minden interfésznek egy felelőssége legyen.

# Interface Segregation Principle

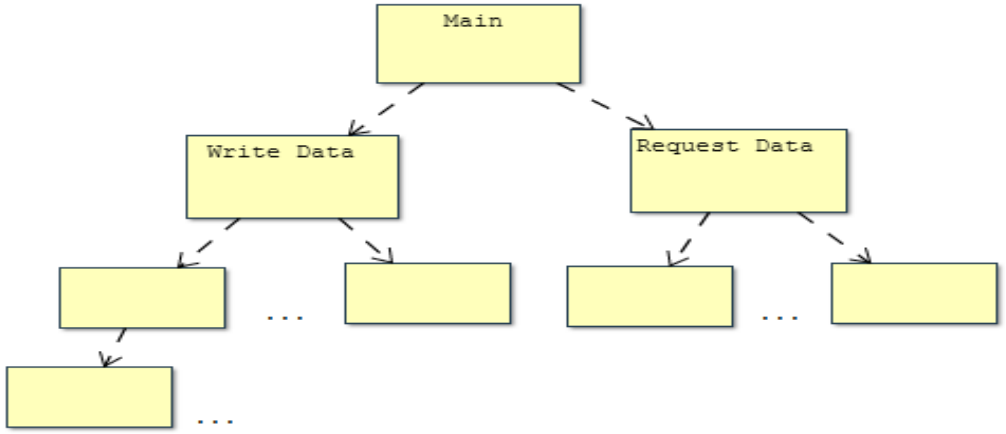
- Az elv megszegése:
  - Felesleges függvények hívása a kívánt függvény előtt
  - Nem használt objektumok átadása konstruktor paraméterként
  - Komplex adat struktúra létrehozása csak egy teszt futtatása céljából
  - Teljes webszerver létrehozása login céljából csak egy egyszerű üzleti szabály
  - *Jar* letöltése csak egy másik *jar* függésének kielégítésére
  - Teszt készítése, amely végigvisz a login folyamaton csak azért, hogy egy üzleti szabályt teszteljünk

# Dependency Inversion Principle

- Az OO tervezés alapja ez az elv
- Mit jelent a függőség?
  - Futás idejű függőségek
    - Amikor a vezérlés futás alatt egyik modult elhagyva a másikba lép be
    - Akkor beszélünk futás idejű függőségről, ha két modul kapcsolatba kerül egymással futás közben
  - Fordítás idejű függőségek (forrás kód függőség)
    - Egy név (pl. osztálynév) definiálva van egyik modulban és megjelenik egy másik modulban is
    - Java-ban: Az A osztály függ B osztálytól, ha az A fordításakor a fordítónak meg kell ismernie B osztályt -> nem fordítható le A osztály B lefordítása előtt

# Dependency Inversion Principle

- Strukturált architektúra



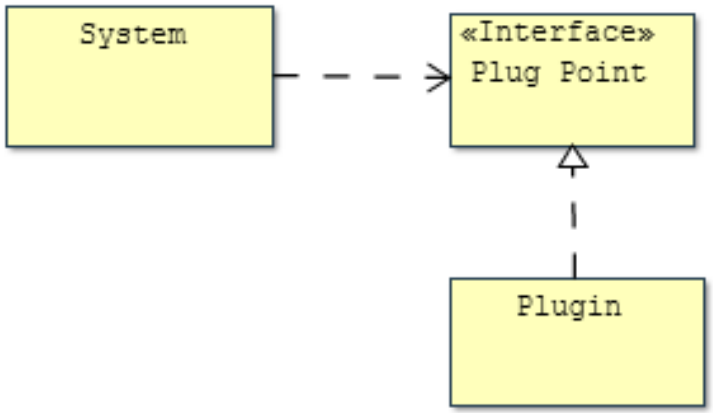

  - Fentről lefele építkező elvet követ
  - A forráskód függőség ugyanolyan irányú, mint a futási függőséget
- Következtetés:
  - Az ilyen típusú rendszerek esetén rossz a függőségek tervezése
  - Miért?

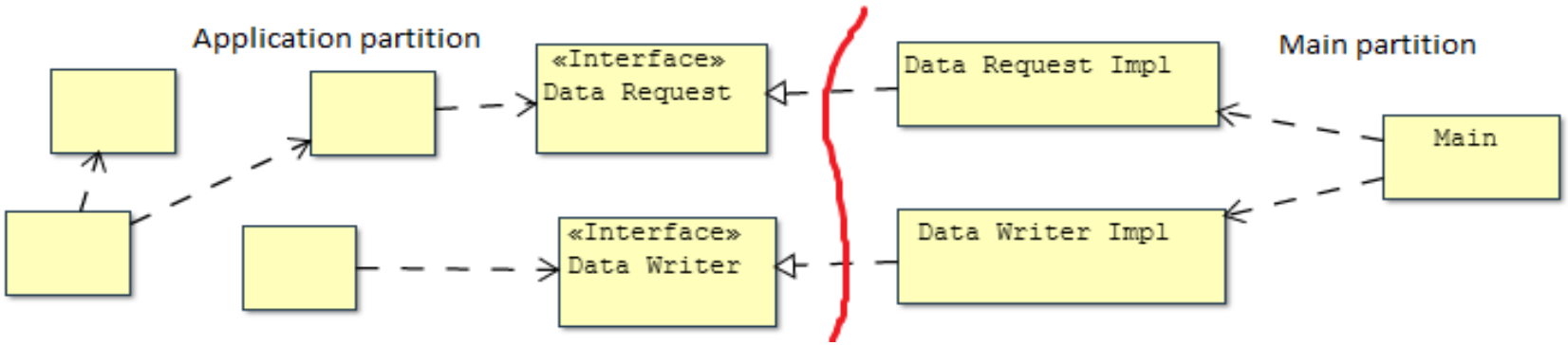
# Dependency Inversion Principle

- Definíció: **a magas szintű logika nem szabad függjön az alacsony szintű részletekről, hanem fordítva**
- Értelmezés: A felhasználási eset nem függhet a technikai megvalósítástól
- A cél, hogy megfordítsuk a függőség irányát a felhasználási esetek és az alacsony szintű részletek között. (pl.: az adatbázis az csak egy plugin-je a felhasználási esetnek)

# Dependency Inversion Principle

- Plugins
  - Modulok közti határok képzésével lehet őket létrehozni
  - A „main” egy plugin-ja a rendszernek (le lehet cserélni, tehát a main csak egy felhasználója az implementációnak/ rendszernek)
  - A UI és a DB réteg is csak plugin
  - Egy jó alkalmazás architektúra követi a plugin architektúrát (újrafelhasználhatóság)

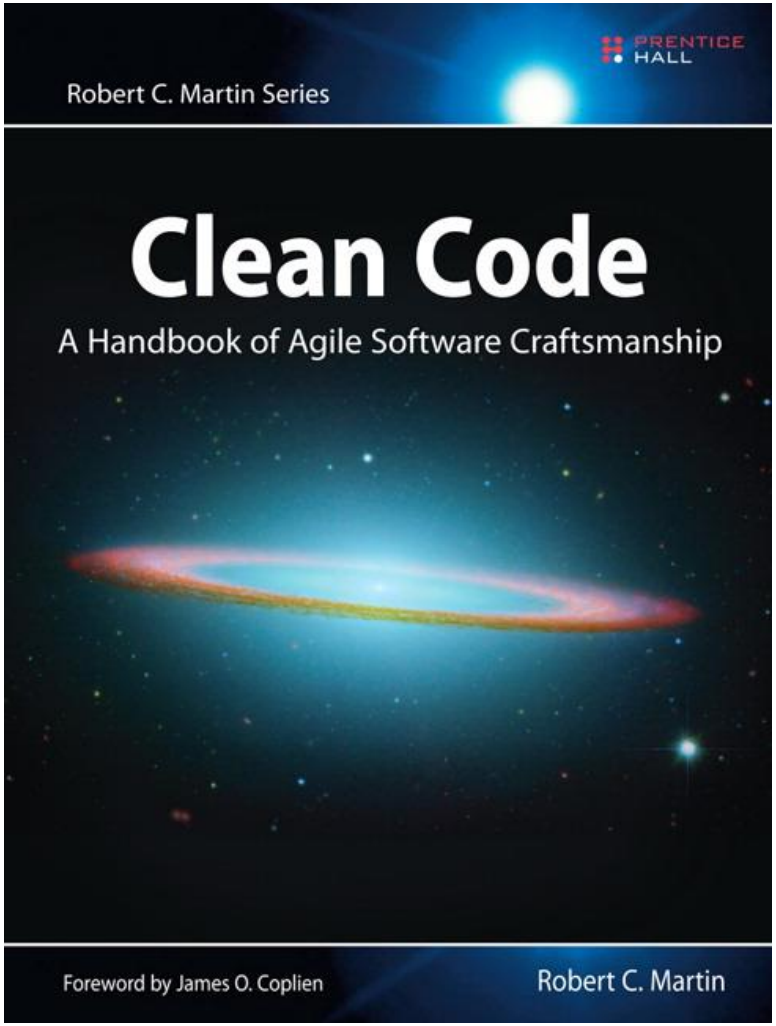


# Clean Code

# Irodalom

- Minden programozónak kötelező olvasmány Uncle Bob könyve!



# Tartalom

- Elnevezési szabályok
- Kommentek
- Formázás
- Függvények
- Refaktorolás

# Mi a Clean Code? Miért van rá szükség?

- Mi a szoftver igazi értéke?
  - Karbantarthatóság
  - Folyamatos továbbfejleszthetőség és szállítás
- Új szoftver rendszerek fő jellemzői:
  - Folyamatosan változó követelmények
- Agilis fejlesztés
- Csapatmunka
- Kódminőség
- Szoftver életciklus

# Kódolási megfontolások

- Osztályok, függvények mérete
- Elnevezés
- Csomagok elnevezése
- Kommentezés (kell egyáltalán?)
  - Kikommentezett kódrészek?
- Egységes formázás
- Kódolási konvenciók

# Elnevezési szabályok

# Mit jelöl a `returnValue` változó?

```
@Override
public int getItemPosition(Object object) {
    int returnValue = POSITION_NONE;
    OfferFragment currentOfferFragment = (OfferFragment) object;
    if (currentOfferFragment.offer != null && OffersSingleton
            .getOfferById(currentOfferFragment.offer.getId()) != null) {
        Integer offerIndex = getOfferIndex(currentOfferFragment);
        if (offerIndex != null) {
            int currentItemPosition = super.getItemPosition(object);
            if (currentItemPosition > offerIndex) {
                returnValue = currentItemPosition - 1;
            }
        }L
    }
    return returnValue;
}
```

# Miért fontosak a nevek a forráskódban?

“Source code is a form of expression and so quality-of-expression must be a primary concern. Poor quality-of-expression creates ongoing costs and impacts the productivity (hence the reputation) of the team that produces it. Therefore software authors must learn to clearly express the intention of variables, methods, classes, and modules. Furthermore, maintenance errors occur when a programmer does not correctly understand the code he must modify.“

*Tim Ottinger cikkjéből: Rules for Variable and Class Naming (1997)*

# Tartsd észben!

1. A nevek az elsődleges eszközei a szándékaink kifejezésére
2. **“Clean code reads like well-written prose.”** - *Grady Booch Object-Oriented Analysis and Design with Applications* című könyéből -> A célunk az kell legyen, hogy olyan kódot írjunk, ami úgy olvasható, mintha mondatokat olvasnánk
3. A könnyen olvasható és megérhető kódot könnyebb karbantartani is
4. A mai modern refaktorolási eszközök mellett már nincs bocsánat arra, hogy nem jó, nem kifejező neveket hagyunk a kódban

# Használj szándékot kifejező neveket!

```
int n; //number of products
```

Ez a comment egy magyarázkodás a nem kifejező név miatt

```
int nrOfProducts;
int numberOfProducts;
```

-> nincs szükség kommentekre, ha kifejező az elnevezés

# Ne informálj félre a nevekkel!

- Félre informálás: egy változó nem azt jelöli, amit a neve mond

```
public class Pair{
    String first;
    String second;
    String third;
}
```

  - Nevezd át az osztályt, metódust, változót, stb. amikor a jelentése változik
  - A kisbetűs L és nagybetűs O használata is félreinformálhat, mert úgy néznek ki, mint a 0 és az 1

```
int O;
int l;
```

# Használj kiejthető változó neveket

- `PG_HUGH`, `PG_HUGA` – Hogy beszélnél ezekről a változókról a kollégáddal pl. telefonon?
- Más pl:

```
public int getYYYY(){
    return this.year;
}
```

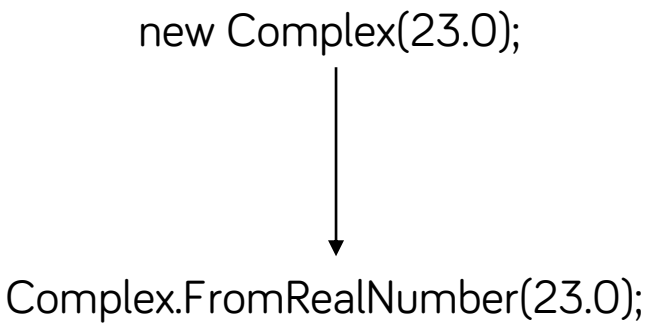
- Kiejthető neveket adj, hogy lehessen azokat egyértelmű módon használni beszédben is!

# Tégy értelmes különbséget

- Kerüld az általános zajszavakat mint Info, Data, Object
  - Használd egyszerűen a `Product` -ot a `ProductInfo`, `ProductData` helyett
  - Használd egyszerűen a `Person` -t a `PersonObject` helyett
- Használd a `destination` és a `source` -t a `string1` és `string2` helyett
- A “variable” szó soha ne szerepeljen a változó nevében
- A változó típusa se szerepeljen a változó nevében
  - Pl: használd a `firstName` -et a `firstNameString` helyett

# Használj főnev és ige szerkezeteket!

- Az osztályok és objektumok nevei főnév vagy főnév szerkezetek legyenek
  - Pl: RequestSender, Product, nrOfProducts
  - Kerüld: …Manager, …Processor, …Data, …Info; ezek túl általánosak
- A metódusok (mutators és accessors is) nevei ige vagy ige szerkezetek legyenek
  - Pl: mutator-ok: calculateDistance(); send(Message message);
  - Ex: accessor-ok: getAge(); setAge(int age);
- A tulajdonságok (properties pl C#-ban) főnevek legyenek
- Preferáld az objektum létrehozó metódusokat az ujabb túlterhelt konstruktorokkal szemben



# Boolean változók elnevezése

- Állítmányos szerkezeteket használjunk névként, hogy az if szerkezettel mondatként legyenek olvashatóak
- Pl:

```
if (isEmpty) {
    showErrorMessage();
}
```

# Enum-ok elnevezése

```
public enum Priority{
    LOW, MEDIUM, HIGH;
}

public enum State{
    STARTED, STOPPED, CANCELLED;
}
```

- Az enumoknak általában leíró vagy állapotjelölő jelentésük van -> a neveik melléknevek legyenek!

# A scope hosszúság szabály

- Változó nevek esetében
  - Kis scope – rövid változó nevek
  - Big scope – hosszú változó nevek
- In case of class and method names
  - Kis scope – hosszú név
  - Nagy scope – rövid név
- A publikus, gyakran használt metódusok kapjanak rövid nevet!
  - Mert a rövid neveket könnyű megjegyezni és kényelmes használni
- A hosszú metódus neveket kis scope-ban használjuk, ilyenkor ezek dokumentációként is szolgálnak
- A kis scope-ú, privát belső osztályok neve lehet hosszú, hogy minél jobban kifejezze az osztály feladatát
- A publikus osztályok kapjanak rövid nevet, kivétel: a származtatott osztályoknak hosszabb neve kell legyen, mint az ősosztályoknak, mivel általában a leszármazott osztály neve az ősosztály neve egy melléknévvel, szóval kiegészítve (`Request, ProductsRequest, Player, TvInputPlayer`)

# Ciklus számlálók és változók nevei

```
for(int i=1; i<11; i++){
    //max 3-4 lines of code
}
```

- Egybetűs ciklus számláló nevek csak a kis scope-ú ciklusokban megengeettek
- `i, j, k` hagyományos ciklus számláló változó nevek

```
for (String s : suffixes) {
    //max 3-4 lines of code
}
```

- Egybetűs ciklus változó nevek csak a kis scope-ú ciklusokban megengeettek

# Ugyanazon dologra ugyanazt a szót használjuk egységesen!

- összezavaró, ha projekten belül vegyesen használjuk:
  - `fetch()`, `retrieve()` és `get()` mint ugyanolyan feldatkörű metódusok más-más osztályokban egy projekten belül
  - egy …`Controller`, egy …`Manager` és egy …`Driver` ugyanabban a folyamatban
  - stb.

# A megoldási terület szakszavai és a probléma terület szakszavai

- Megoldási terület szakszavai
  - Algoritmus, tervezési minta, matematikai fogalmak nevei
  - Pl: Observer, Factory, BinaryTree
  - Alacsony absztrakciós szintű kódban használjuk őket!
- Probléma terület szakszavai
  - Annak a területnek a szakszavai, amelyre a szoftver készül
  - Pl. Egy bérszámfejtő alkalmazás esetében: payroll, employee, account
  - Magas absztrakciós szintű kódban használjuk őket!

# A rossz nevekről

1. Hogyan tudod tesztelni, hogy a neveid jók? Írj unit teszteket, amelyek használják ezeket a neveket!
2. Ha meg kell nézned az implementációs részleteket ahhoz, hogy megérts egy nevet, akkor az rossz név!
3. Ha magyarázó kommentet kell írnod egy név mellé, akkor nem elég jó nevet választottál!

# Bibliográfia


- Tim Ottinger: **Rules for Variable and Class Naming (1997)**
- Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin
- Clean Code Videó Sorozat- http://cleancoders.com/
- https://en.wikipedia.org/wiki/Hungarian_notation

# Kommentek

# Ritkán írjunk kommenteket!

- Ha sűrűn jelennek meg kommentek a kódban, elkezdünk nem törődni velük
- Tartogasd a kommentjeidet olyan speciális esetekre, amikor a figyelem felhívására van szükség
- A jó komment olvasója hálás kell legyen, hogy az a komment megíródott

# A kötelező, erőltetett kommentek

- Amikor a fejlesztők kötelezve vannak arra, hogy kommenteket írjanak, akkor ezek a kommentek
  - gyengék,
  - semmitmondóak,
  - jelentéktelenek lesznek
  - és könnyedén elavulttá válnak

# A komment, mint kudarc

- Amikor a kód jól kifejezi a programozó szándékát, akkor nincs szükség kommentekre
- A komment írás sok esetben az önkifejező kód írásának kudarca

```
int n; //number of products
```

- A modern nyelveknek, mint a Java, C#, Ruby jó névválasztással olyan kifejező erejük van, hogy kommentekre nincs szükség
- Minden komment, amit leírsz, egy kudarc, hogy nem tudtad kódban kifejezni magadat

# Nyelvek, ahol szükségesek a kommentek

- Nem minden nyelvnek jó a kifejező ereje
  - Pl: Assembly-ben sűrűn ajánlott kommentezni, hogy a kódot könnyebben értse egy olvasó
- Korábban, amikor még az erőforrás megszorítások domináltak a programok struktúrája fölött, a kommentek hasznosak voltak

# Komment “rothadás”

- Lehetséges okok:
  - Nem lokális kommentek: egy feature-höz tartozó kommentek nemcsak az implementáló forráskód mellett találhatóak meg, hanem pl az osztály elején is, vagy máshol a modulban
  - A programozók sokszor elfelejtik a kommenteket frissíteni, amikor a hozzájuk tartozó kódot módosítják

# Jogi kommentek

```
/*
 * Copyright (c) 1997, 2014,
Xcompany and/or its affiliates. All
rights reserved.
*/

```

- Ott kell legyenek…

# Informatív kommentek

- Valóban hasznos információt nyújtanak az olvasónak, aki hálás is lesz ezért

```
//matches any alphabetic character
between m and n
String idPattern = "m[a-z]n";
```

# Tisztázások és szándék magyarázatok

- Amikor tisztázni kell dolgokat vagy magyarázkodni kell kommentben, ismét egy kudarcon vagy túl…

```
/**
 *
 * The implementation of the algorithm
is not the best. Please correct it if
you can.
 *
 */
```

# Következménykre való figyelem felhívás

```
/**
 * Do not pass i > 3000 value
 */
private void foo(int i){
  //...
}
```

- Gyakran ilyenkor arról van szó, hogy valamit nem sikerült megfelelően implementálni

# TODO kommentek

```
//TODO check that the password is
not empty
```

- Igyekezz nem hagyni TODO kommentek a becsekkolt kódban! Végezd el a feladatot inkább mielőtt becsekkolsz!

# Publikus API documentáció

```
/**
* Returns the number of elements in this list.  If this list
contains
* more than <tt>Integer.MAX_VALUE</tt> elements, returns
* <tt>Integer.MAX_VALUE</tt>.
*
* @return the number of elements in this list
*/
int size();
```

- Néha nagyon hasznosak tudnak lenni
- “The best public API documentation is the documentation that you don’t have to write” – Uncle Bob
- Legtöbb esetben nem szükségesek az ilyen kommentek, ha jók az elnevezések

# Mormogások

```
// after a long discussion with
team members, we decided to use the
Observer pattern


// Hello, please maintain me!
```

- Csak forráskódot írj a forráskód fájlokba!

# Redundáns magyarázatok

```
/**
* The id of this campaign.
*/
private long campaignId;
```

- Semmi szükség a nevek megismétlésére kommentekben…
- A jó komment mindig valami új információval szolgál
- Emlékezz a DRY elvre: Don’t repeat yourself!

# Kötelező redundancia

```
    /**
     * Converts a domain user to Spring Security specific user.
     *
     * @param user
     * @return
     */
    protected UserDetails convertToSpringUser(User user) {
        return new
org.springframework.security.core.userdetails.User(user.getUsern
ame(), user.getPassword(),
Collections.singleton(convertRoleToSpringAuthority(user.getRole(
))));
    }
```

- "Maybe they should be called mandated stupidity" - Uncle Bob
- "All we are saying is give code a chance!" - Uncle Bob

# Naplózási kommentek

```
//10.03.2015 - initial version
//12.03.2015 - search implemented
//13.04.2015 - search fixed
//13.06.2015 - getElements method
added
```

- A verzió kezelést hagyjuk a verzió követő rendszerre!

# Zaj-kommentek

```
/**
 * Default constructor.
 */
public Product(){
}
```

- Töröljük őket!

# Sáv-kommentek

```
// ********************************

// Private attributes

// ********************************


private String regexp;


// ********************************

// Default constructor

// ********************************

public MessageParser() {


}
```

```
// ****************************************
// Public methods
// ****************************************
public void parse() {

}

// ****************************************
// Private methods
// ****************************************
private List<String> splitIntoParts() {

}
```

- “Nothing shouts ‘Ignore me!’ more than a big banner comment!” – Uncle Bob

# Bezáró zárójel kommentek

```
if(condition1){


}//if
while(condition2){


}//while
```

- Ma már a modern fejlesztői környezetekben nincs rájuk szükség

# Szerző neve, beceneve kommentben

```
/**
 *
 * @author laci
 */
```

- A verzió követő rendszer a fájlok szerzőit is tárolja, ilyen kommentekre semmi szükség

# HTML a kommentben

```
/**
* Returns <tt>true</tt> if this list contains the specified element.
* More formally, returns <tt>true</tt> if and only if this list contains
* at least one element <tt>e</tt> such that
* <tt>(o==null ? e==null : o.equals(e))</tt>.
*
* @param o element whose presence in this list is to be tested
* @return <tt>true</tt> if this list contains the specified element
* @throws ClassCastException if the type of the specified element
*         is incompatible with this list
* (<a href="Collection.html#optional-restrictions">optional</a>)
* @throws NullPointerException if the specified element is null and this
*         list does not permit null elements
* (<a href="Collection.html#optional-restrictions">optional</a>)
*/
boolean contains(Object o);
```

- Zavarossá teszik a kommentet…

# Nem lokális kommentek

- Könnyen elavulttá válnak idővel
- Ne írjunk kommentet egy más helyen levő kódról, a komment mindig a mellette levő kódhoz kapcsolódjon

```
/**
* Parses a message into more parts. These parts will be used and
shown by the Products activity.
*
*/
public class MessageParser{
}
```

# Kikommentezett kód

```
// public DefaultTokenServices tokenServices() {
// DefaultTokenServices tokenServices = new DefaultTokenServices();
// // tokenServices.setSupportRefreshToken(true);
// // tokenServices.setTokenStore(this.tokenStore);
// tokenServices.setAccessTokenValiditySeconds(accessTokenValiditySeconds);
// return tokenServices;
// }
```

- Ezek a kódok már megvannak a verzió követő rendszerben
- Töröld őket anélkül, hogy elolvasnád!

# Tartsd észben!

- A változó, osztály, metódus, modul neveide legyen kifejfezőek ahelyett, hogy kommenteket írj hozzájuk magyarázatként
- “Make your code read like well written prose!” – Uncle Bob

# Bibliográfia


- Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin
- Clean Code Videó Sorozat - http://cleancoders.com/

# Formázás

# Fontos a formázás?

- Amikor a forráskódod olvasója ránéz a kódodra, akkor először akaratlanul is annak formázását figyeli meg (és nem a neveket vagy valami mást)
- A formázás a kommunikációról szól
- Egy jól “kommunikáló” kódot könnyű változtatni és karban tartani, tehát a cégnek pénzt takarít meg

# Mi számít a formázásnál?

- Fájl méret
- Függőleges formázás
- Vízszintes formázás
- Indentáció

# Fájl méret

- Rövid fájlméreteket tarts fenn!
- De mi számít rövidnek?
  - A statisztikák arra mutatnak rá, hogy azok a szoftverek, amelyek TDD-vel készültek, átlagosan 50-60 soros fájlokat tartalmaznak, míg más szoftverek 200-at

# Függőleges formázás – Üres sorok elválasztó eszközként

- Válasszuk el a metódusokat egymástól
- Válasszuk el a privát és publikus attribútumokat egymástól

```
public class A {

    private String a;
    private int b;

    public static final String c;

    public A() {

    }

    public void foo() {

    }

    public void bar() {

    }
}
```

# Függőleges formázás – Üres sorok elválasztó eszközként

- Azok a dolgok, amelyek egymáshoz kapcsolódnak legyenek vertikálisan közel egymáshoz
- Azok a változók, amelyek nem kapcsolódnak szorosan egymáshoz legyenek elválasztve egy üres sorral

```
private ActionBarDrawerToggle actionBarDrawerToggle;
private List<DrawerMenuItem> navigationDrawerItems;

private Offer currentOffer;

private int selectedItemPosition = -1;
private boolean canRemoveSelections = true;
```

# Függőleges formázás – Üres sorok elválasztó eszközként

- Válszald el üres sorral az egy metóduson belüli kódblokkat (kis metódusokban erre ritkán lesz szükség ☺)

```
@Override
public Object instantiateItem(ViewGroup container, int position) {
    View itemView = layoutInflater.inflate(R.layout.li_global_tutorial_pager_item, container, false);

    ImageView imageView = (ImageView) itemView.findViewById(R.id.ivGlobalTutorial);
    imageView.setImageResource(mResources[position]);

    container.addView(itemView);

    return itemView;
}
```

# Függőleges formázás – Üres sorok elválasztó eszközként

- Válaszd el a “given”, “when”, “then” blokkokat a tesztekben

```
@Test
public void requestSenderService_shouldBeStarted() {
    //given
    fillOutTheUsernameEditTextWith("John");

    //when
    activityUnderTest.findViewById(R.id.down).callOnClick();

    //then
    Intent startedService = Shadows.shadowOf(activityUnderTest).getNextStartedService();
    assertThat(startedService.getComponent().getClassName()).isEqualTo(HttpPostRequest.class.getName());
}
```

# Vízszintes formázás – sorok hossza

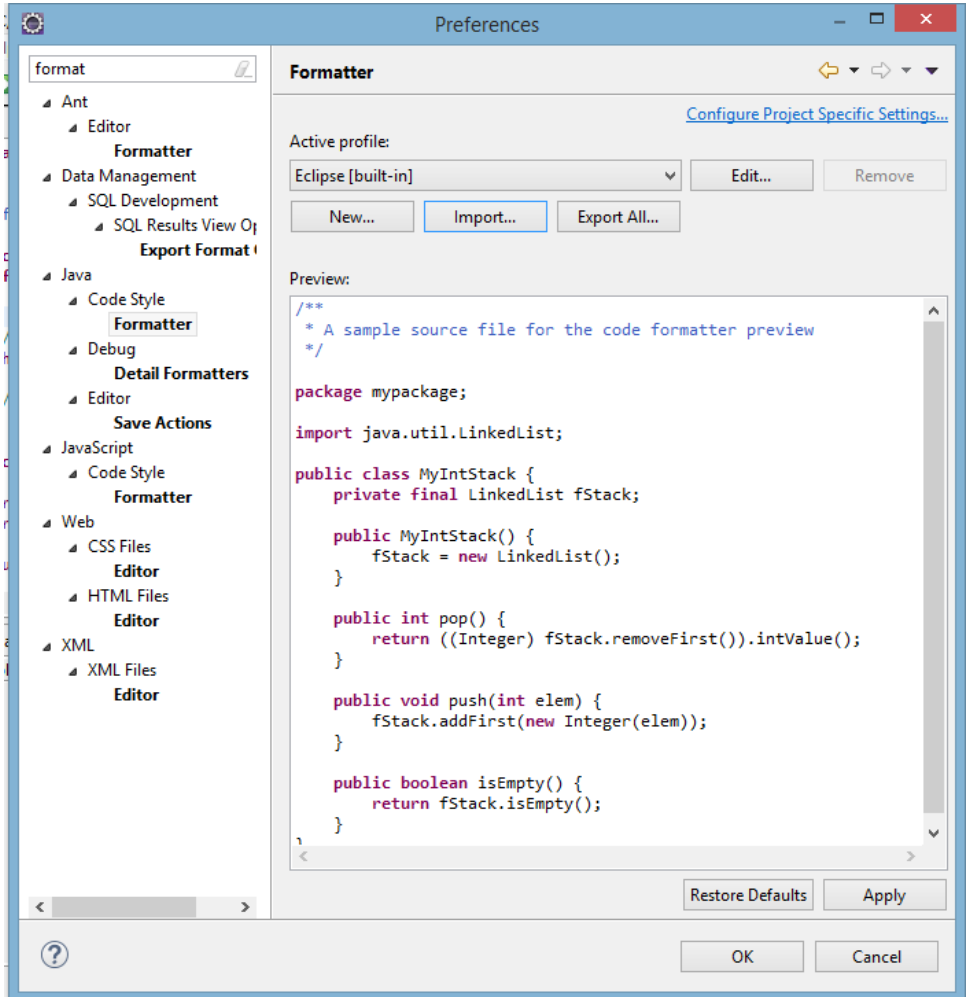
- A megfelelő sorhosszúság az, amikor nem kell jobbra szkrolloznunk, hogy lássuk a sor végét
  - > De a képernyőméretek folyamatosan nőnek…
- Irányadó szabály: a sorok hossza ne haladja meg a 100-120 karaktert

# Indentáció

- Szóközök és tabok is jók indentációra
- 2,4 vagy akár 8 karakter hosszú indentációt is használhatunk
- De: használja mindenki ugyanazt az indentációs stílust a csapatban

# Kód formázási standard a csapatban

- Az indentációt nézve egy kód olvasónak nem szabad ráismernie a kód szerzőjére a csapatban
- Ne formázd a saját ízlésedre a kódot kicsekkoláskor, majd vissza a csapat stílusára becsekkoláskor
- A csapatban mindenkinek ugyanazokat a kód formázási szabályokat kell alkalmaznia
- Adjatok meg kód formázási szabályokat a fejlesztő környezetetnek egy konfigurációs fájl formájában, használja ugyanazt a fájlt az egész csapat



# Bibliográfia


- Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin
- Clean Code Videó Sorozat - http://cleancoders.com/

# Függvények

# Egy függvény egy és csakis egy dolgot csináljon

- De mi az az egy dolog?
  - Nézőponttól függ…
  - De: Ha addig refaktorolod a függvényt, amíg több új függvénybe nem tudod kiszervezni a kód blokkokat, akkor tudhatod, hogy már csak egy dolgot csinál a függvényed ("extract till you drop")
- A függvényen belül csak egy absztrakciós szintű kód szerepeljen
  - Egyszeri indentálás legyen befele minden sorban
  - Ne legyenek belső kapcsos zárójelek

# Milyen hosszú legyen egy függvény?

- A hagyományos szabály, miszerint egy képernyőnyi hosszúságú legyen egy metódus már nem alkalmazható a nagy képernyőméretek miatt
- 4-6 sor ideális hosszúság, 10 sor elfogadhatatlan
  - Az if, switch, for, while, try-catch, stb. blokkok 1 sorosak legyenek kapcsos zárójelek nélkül
- Domborzat metafóra: a metódusok az osztályon belül domb-síkság váltakozások formáját kell adják, hogy könnyen kereshető, navigálható és olvasható legyen az osztály
- Az osztályok sokszor a hosszú függvényekben bújnak meg
  - Az *Extract Method Object* (*Kiszervezés Metódus Objektumba*) refaktorálási techinkával a hosszú föggvényekől szervezzünk ki egy osztályt

# Hogyan nevezzük el a függvényeket?

- Igét vagy ige szerkezetet válasszunk
- Az alacsony absztrakciós szintű metódusoknak hosszú, leíró neveik legyenek
  - Pl: `checkForAvailableUpdates();`
- A magas absztrakciós szintű, publikus metódusoknak rövid, tömör, megjegyezhető, kényelmesen használható neveik legyenek
  - Pl: `send(Request request)`

    és nem `sendOrThrowIoException(Request request)`

# Argumentumok száma

```
saveUser(int id, Role role, String name, int age)
```

- A függvény neve és az argumentumai együttesen intuitiv, tiszta és olyan olvasható kell legyen, mint egy mondat
- Háromnál több argumentum nehezen olvasható, nehezíti a függvény használatát és nehezen megjegyezhető
- Maximum hány argumentuma legyen egy függvénynek?
  - **max 3 argumentum** legyen, ez üres argumentum lista a legideálisabb
- De mit tegyünk, ha háromnál több paramétert kell átadnunk?
  - Írjunk egy új osztályt, amiben összefogjuk a paramétereket!
    - De ezzel nem sokat oldalttunk meg, mert ennek a példányát egy sok paraméteres konstruktorral kellene létrehozni…
    - Megoldás: sok paraméteres konstruktor helyett setter metóduskat használjunk
    - Kötetkezmény: egy kis ideig az új objektumunk félig kész állapotba lesz; a unit tesztekkel biztosítanunk kell, hogy ilyen állaptó objektumokat nem használjunk sehol

# Boolean paraméterek

```
sort(true);
```
where `public void sort(boolean ascending);`

- A **`boolean`** paraméterek sok esetben hiba forrásokká válnak, mert összezavaróak
- Legtöbb esetben egy **`boolean`** paramétert váró függvény két dolgot csinál: egyet a **`true`** esetre, egyet pedig a **`false`** esetre
  - megoldás: írj két függvényt, egyet a **`true`** estre, egy másikat a **`false`** esetre

```
sortAscending();
sortDescending();
```

- Két boolean paraméterű függvény még rosszabb eset: ez 4 esetet kezel
- NE ÍRJ BOOLEAN PARAMÉTERT VÁRÓ FÜGGVÉNYT!
  - Írj két függvényt! Egyet a true esetre, egy másikat a false esetre!

# Null érték átadása paraméterként és a “null védekezés” problémája

- Ugyanaz a probléma, mint a `boolean` paraméterek esetében, a függvény két dolgot csinál: egyet a `null` értékre, egyet a nem `null` értékre
- Védekező programozás (paraméter null vizsgálata a függvény elején) szükséges a publikus API-kban, de csapatban a projekt belső API-jának fejlesztésekor nem
- A védekező programozás egy jel, hogy nem bízol a csapattagjaidban
  - A legjobb védekezés a támdás, a legjobb támadás pedig egy jó teszt gyűjtemény
- a boolean és null paraméterek olvashatatlanná teszik a kódot a függvényhívás helyén
- De mi van akkor, ha egy olyan külső API-t használunk, ami null vagy boolean paramétereket vár?
  - Szervezd ki ezeket ez értékeket jól elnevezett konstansokban, hogy a kód olvasható legyen
  - pl: a `send(query, null);` helyet `send(query, NO_PARAMETERS);`
    ahol `private static boolean NO_PARAMETERS = null;`

# A bemenő paraméterek ne legyenek kimenő paraméterek!

```
static boolean getCookieValue(String aCookieName, HttpServletRequest
aRequest, final StringBuilder aCookieValue) {
    if (aCookieValue.length() > 0)
        throw new IllegalArgumentException();
    boolean userHasCookie = false;
    Cookie[] cookies = aRequest.getCookies();
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if (aCookieName.equals(cookie.getName())) {
                userHasCookie = true;
                // change the state of the output param (aCookieValue)
                aCookieValue.append(cookie.getValue());
            }
        }
    }
    return userHasCookie;
}
```

# A bemenő paraméterek ne legyenek kimenő paraméterek!

- Az emberek nem számítanak arra, hogy a bemenő paramétereken keresztül adat jöjjön vissza
  - Amikor egy olvasó lát egy bemenő paramétert, azt várja el, hogy azon keresztül a függvény adatok kap és nem adatot ad
  - A kimenő paraméterek zavart okoznak az olvasónál és dupla effort szükséges hozzájuk
- **NE HASZNÁLJUNK KIMENŐ PARAMÉTEREKET!**
- A kimenő adatot a `return` blokkon keresztül térítsük vissza!

# A stepdown szabály – az újság hasonlat

- Hogyan vannak a címek és bekezdések rendezve egy újságban?
  - A fontos dolgok a lap tetején vannak, a kevésbé fontosan, a részletek pedig alul
  - Az egymást követő bekezdések mind részletesebbek és részletesebbek
- Két előny:
    - A szerkesztő levághatja a cikk végét, nincs elég hely anélkül, hogy a lényeg elveszne
    - Az olvasó felülről kezdheti az olvasást és addig folytatja levelefe, amíg meg nem unja

# A stepdown szabály

```
public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        firstChildOfFirstPublic();
        secondChildOfFirstPublic();
    }

    private void firstChildOfFirstPublic() {
        firstChildOfFirstChildOfFirstPublic();
    }

    private void firstChildOfFirstChildOfFirstPublic() {
    }

    private void secondChildOfFirstPublic() {
        firstChildOfFirstChildOfSecondPublic();
    }

    private void firstChildOfFirstChildOfSecondPublic() {
    }

    public void secondPublicMethod(){
        firstChildOfSecondPublic();
    }


    private void firstChildOfSecondPublic() {
    }
}
```

- lefele haladva az osztályban, a metódusok absztrakciós szintje csökken
- Nincsenek visszautalások, nem kell felfele keresgélni
- A második publikus, magas absztrakciós szintű metódus az első publikus függvény által meghívott összes függvény után jön
- Az is elfogadott elv, ha előbb az összes publikus metódus szerepel az osztályban, és csak utánuk a kis absztrakciós szintűek

# Mellékhatások (side effects)

- Mikor beszélhetünk mellékhatásokról?
  - Amikor egy metódus megváltozatja egy olyan változó értékét, ami rajta kívül él
  - Ez nehézséget okozhat a kód megértésében és a hibakeresésben
  - Sokszor hibák forrásai

# Páros függvények (Temporal coupling)

- Metódusok melyeknek mellékhatásaik vannak

és egymás után, megadott sorrendben kell őket meghívni: `open -> close, new -> delete, set -> get`

- Sokszor hibaforrás, hogy a párnak ez egyik felét nem hívja meg a programozó

# Páros függvények – hibák elkerülése

- Sok esetben a hibákat ki lehet küszöbölni
  - Egy ilyen módszer a blokk átadás (“passing a block”)

```
private void open(File file, FileCommand fileCommand){
    file.open();
    fileCommand.process(file);
    file.close();
}
```

    - Így garantálva van a jó sorrend
    - De ez sem tökéletes megoldás, maradnak mellékhatások
- Nem szeretnénk megszabadulni minden mellékhatástól, hiszen:
  - Akarunk adatbázisban menteni
  - Fájlokat írni
  - Stb.
- A cél, hogy kontrolláljuk, ahogyan a mellékhatások történnek, amire egy jó technika a CQS

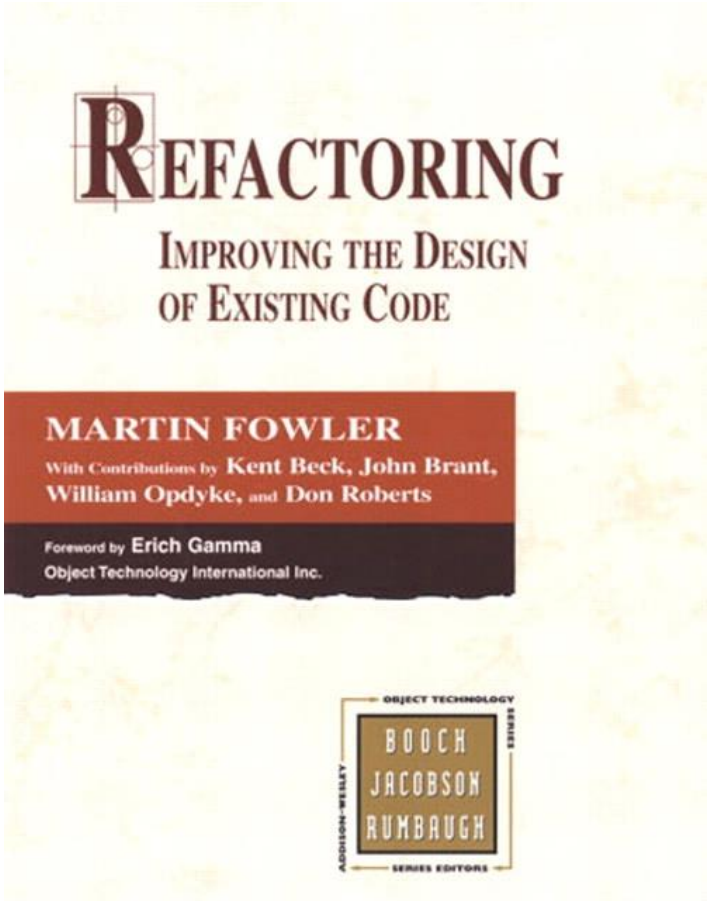
# Kérések és utasítások definiálása

- Kérés
  - Nem változtatják a rendszer állapotát
  - Pl: `private String getContent(File file)`
  - Értéket térítenek vissza, mely vagy egy számítás eredménye, vagy a rendszer
- Utasítás
  - Változtatja a rendszer állapotát, azaz van mellékhatás
  - Pl: `private void save(String username)`
  - Nincs visszatérítési értéke
    - De akkor, hogy jelezzük, ha hiba lépett fel, ha nem téríthet vissza sem `true/false`-t, sem `null`-t?
    - Hiba esetén dobjon kivételt!
- Tipikusan a getterek kérések, a setterek utasítások

# Refaktorálás

# Bibliográfia

- Értékes olvasmány (13 éve készült, de még mindig aktuális!)



# Mit jelent a refaktorálás?

- “Meglévő kód architektúrájának javítása”
- Kód struktúra változtatása a funkció változtatása nélkül
- Kód tisztítás, hogy olvashatóbb és karbantarthatóbb legyen
- Cserkész elv követése: „Mindig hagyd tisztábban a tábort, mint ahogy találtad!”
- Hagyományos szoftverfejlesztés: tervezés, majd kódolás és a kód egy idő után elkezd romlani
  - Refaktorálás: rosszul megtervezett szoftvert lehet javítani!
- Martin Flower a könyvében 72 konkrét technikát fogalmaz meg

# Hogyan biztosítható hogy a refaktorálás nem változtatja meg a viselkedést?

- Unit tesztek futtatása (TDD…)
- Főbb szabályok
  - Refaktorálás előtt győződjünk meg róla, hogy az adott kódrész le van fedve tesztekkel!
  - Tesztek futtatása refaktorálás előtt!
  - Tesztek futtatása minden refaktor lépés után!

# Gyakori jelek amikor refaktorálásra van szükség

- Duplikált kód
- Komment
- Rossz nevek
- Hosszú osztályok
- Hosszú függvények
- Middle man probléma (közbeékelt felesleges objektum)
- Feature envy
- Hosszú paraméter lista
- Switch
- Visszautasított örökség
- Stb.

# Duplikált kód

```
Test(expected = Amoba.PositionOccupatedException.class)
public void onePosition_canBeOccupatedOnlyOnce() {
  // given
  Amoba amoba = new Amoba();

@Test
public void nullReturnedAsWinner_whenGameIsNotFinished() throws Exception {
  // given
  Amoba amoba = new Amoba();
}
```

Duplikált kód eltávolítása az **Extract Method** technikával, majd a belső változó **kiszervezése mezővé**.

```
private Amoba amoba;

@Before
public void setup() {
  amoba = new Amoba();
}
```

# Kommentek

```
//uploads the file to the server, throws exception
if not succeeded
private void send(File file) throws IOException {


}
```

Minden függvény, változó, stb átnevezése ami magyarázatra szorulna.

```
private void tryToUpload(File file) throws
IOException {


}
```

# Rossz nevek

- `GameTest` osztály, ami tartalmazza az `Amoba` osztály tesztjeit
- `AmobaTest` név sokkal jobb lenne.
- **Ne féljünk átnevezni az osztályokat, függvényeket, változókat, packageket**

# Hosszú függvények, hosszú osztályok (Spagetti kód)

- A példa nem férne ki erre az ablakra☺
- Az ilyen kód tipikusan megszegi a Single Responsibility elvet
- Szedjük szét a hosszú függvényeket rövid függvényekre, szükség esetén külön osztályba
- A hosszú függvények tipikusan önálló osztályokat rejtenek: **Extract Method Object** technika
  - A hosszú függvény belső változói az új osztály mezői lesznek
  - A függvényben levő kód blokkok az új osztály kis metódusai lesznek
- Hosszú osztályok több dologért felelősek
  - Válasszuk szét az osztályt több kisebb osztályra, mígnem csak egy felelősségük lesz

# Middle man probléma

- Middle man probléma: egy metódus hívás valójában csak áthív egy másik objektum egy függvényére

```
amoba.playerX.draw(amoba, 0, 0);
```

Player osztály:

```
public void draw(Amoba amoba, int
xPosition, int yPosition) {
    amoba.draw(sign, xPosition,
yPosition);
}
```

- Middle man eltávolítása: **Remove Middle Man, Inline Method** vagy **Replace Delegation with Inheritance**

# Test Driven Development

# Bevezetés

- Volt már valaha dolgunk rossz kóddal?
  - Olvasni kódot több idő, mint írni
- Volt már olyan eset, hogy nem volt idő egy feladat „szakszerű” megoldására, a kód tisztítására, a rövid határidő miatt?
- Ahogy nő a kód mennyisége, csökken a fejlesztők produktivitása

# Mit értünk TDD alatt?

- Hagyományos fejlesztési ciklus (hosszú ciklus, több hónapos):


- Test-driven development ciklus (ismétlődő rövid ciklusok, pár perc):


- A TDD egy programozási technika, ami a következő gondolaton alapul: csak egy bukó teszt kizöldítésére írjunk production kódot
- A TDD három szabálya:
  1. Írjunk egy elbukó tesztet
  2. Írj csak annyi production kódot, ami kizöldíti a tesztet
  3. Tisztítsd (refactor) a tesztet és a production kódot
- Red-green-refactor

# Összefoglalás

# Cserkész szabály

- Nem elég jól megírni a kódot, hanem tisztán is kell tartani
- „Hagyd a tábort tisztábban, mint ahogy kaptad!”

# Köszönöm a figyelmet!

peter.ekler@aut.bme.hu

balogh.tamas@autsoft.hu

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék